

# Real-time Visual Tracker by Stream Processing

## Simultaneous and Fast 3D Tracking of Multiple Faces in Video Sequences by Using a Particle Filter

Oscar Mateo Lozano · Kazuhiro Otsuka

Received: 12 September 2007 / Revised: 25 April 2008 / Accepted: 12 June 2008 / Published online: 12 July 2008  
© The Author(s) 2008

**Abstract** In this work, we implement a real-time visual tracker that targets the position and 3D pose of objects in video sequences, specifically faces. The use of stream processors for the computations and efficient Sparse-Template-based particle filtering allows us to achieve real-time processing even when tracking multiple objects simultaneously in high-resolution video frames. Stream processing is a relatively new computing paradigm that permits the expression and execution of data-parallel algorithms with great efficiency and minimum effort. Using a GPU (graphics processing unit, a consumer-grade stream processor) and the NVIDIA CUDA™ technology, we can achieve performance improvements as large as ten times compared to a similar CPU-only tracker. At the same time, the Stream processing approach opens the door to other computing devices, like the Cell/BE™ or other multicore CPUs.

**Keywords** Stream processing · GPGPU · CMP · Particle filtering · Video tracking · Real-time systems

### 1 Introduction

Fast and robust object tracking in video sequences is required by many applications in many fields: automated surveillance systems follow their targets using these techniques, robots rely on it to perform navigation tasks or man-machine interaction, augmented reality systems depend on the position data acquired by visual tracking to place their virtual objects in the real world, video-games or assisted devices can be controlled thanks to a camera and some face or hand tracking software, to name just a few.

Our motivation for developing a real-time face video tracker is to advance research on a system [1] that can infer conversation structure from video sequences of face-to-face communication, in which it is assumed that the gaze direction of the participants, which provide cues for discerning the conversation structure, can be identified from head directions. This system could be applied to a wide range of video-based applications such as computer mediated communications. The constraints we impose on this tracker are: it has to be completely automatic, robust against rapid movement and partial occlusion, work with just one camera (no stereo-vision) on a conventional PC, and be able to track several faces simultaneously in real-time.

#### 1.1 Particle Filtering

Particle filtering [2] is a model estimation technique based on Monte Carlo simulations. Random values of a state-space variable are generated (the so-called particles), used in a description of the system, and checked against the current output measure to generate a weight value, or likelihood of that particle being the

---

O. Mateo Lozano · K. Otsuka (✉)  
NTT Communication Science Laboratories,  
3-1, Morinosato-Wakamiya, Atsugi-shi 243-0198, Japan  
e-mail: [otsuka@eye.brl.ntt.co.jp](mailto:otsuka@eye.brl.ntt.co.jp)

O. Mateo Lozano  
e-mail: [omateo@locke.es](mailto:omateo@locke.es)

O. Mateo Lozano  
Image Processing Group, Universidad Politécnica de  
Madrid, Madrid, Spain

one that best describes the current state of the system. Therefore, the collection of all these particles and their weights in each instant is a numerical approximation of the probability density function of the system. The particle filter (PF) framework is the basis of the well-known condensation (conditional density propagation [3]) algorithm, which was originally proposed for contour tracking, but has been also successfully applied to the appearance-based tracking of moving objects in video sequences. The probabilistic approach of these methods provides significant robustness, as several possible states of the system are tracked at any given moment.

A common problem with this kind of practical method is its significant computational requirements: when the number of particles becomes very large, Monte Carlo characterization becomes a better representation of our process, but the costs of the algorithmic calculations also increase. Fortunately, particle filters are easy to parallelize; they require high arithmetic throughput (as opposed to low latency), and have low global communication and storage costs. It is our belief that the advent of consumer-grade parallel processors can bring the robustness of these algorithms to real-time applications.

## 1.2 Stream Processing and GPGPU

We are specially interested in computer graphics chips (known as “graphics processing units” or GPUs), because they are currently the most powerful and cheap computational hardware available [4]. These chips have gone from fixed-application peripherals to modern, powerful, and programmable general purpose processors. In recent years, there has been strong interest from researchers and developers in exploiting the power of commodity graphics hardware for general-purpose computing (this movement is known as GPGPU, for “general purpose GPU”). Unfortunately, the GPU uses an unusual programming model, so effective GPU programming requires rewriting the target algorithm into graphics terms; the programmer must be familiar with the limitations of the underlying hardware.

We also note that other multicore processors (or CMPs, for “chip multiprocessors”), like GPUs, suffer from programming difficulties. The traditional serial programming approach does not take advantage of the additional cores in the processor. The typical thread programming method (which raises the issues of synchronization, deadlock, load balancing, ...) is very hard to implement and does not scale well given the number of cores in current and future multicore CPUs. The Cell

Broadband Engine processor allows many kinds of parallel processing techniques (such as DMA operations) and program scheduling in the “synergistic processing elements” can be done by hand by the programmer. The complexity of this, however, makes this task non-trivial.

The programming paradigm raised by stream computing can be described as, given a set of input and output data (streams), to define some computation-intensive operations (kernel functions) that are to be applied to each element in the stream while exploiting the data independency and locality typical in media-processing applications. The programmer is forced by this (intuitive) programming model to express his/her application in a way that adapts well to the computational resources of CMPs. Many believe that Stream processing represents an important advancement in making parallel processing easily accessible to programmers: NVIDIA Corp. provides now a full architecture based on the Stream processing model called CUDA™ (Compute Unified Device Architecture [5]), and AMD, Inc. has recently released a stream processor solution based on both hardware and a low-level SDK called CTM™ (Close To Metal) [6]. Rapidmind [7] is a software solution that can realize Stream processing on top of the most common CMP architectures, ....

Stream processing offers more than just ease of programming: architectures that map well to this paradigm (“stream processors”) can achieve higher performance than other architectures [8], as the locality and concurrency enforced by this paradigm (and the associated data bandwidth hierarchy) allows more of the die to be devoted to ALUs instead of caching and memory access circuitry. GPUs (especially since the advent of unified architectures that exchange the vertex and pixel shaders found in last-generation devices for generic SIMD processors, albeit with some limitations [9]) or the Cell BE (with appropriate software support) can be considered general-purpose stream processors. Universities like Stanford [10] and companies like SPI [11] are working on pure stream processor implementations.

It is our belief that the particle filter framework applies especially well to the stream processing paradigm: the operations to be performed are very simple (in our case, they consist mainly of geometric transformations), the data to use is highly localized (each particle is self-contained), and needs little memory, so we can use the ability offered by stream processors (perform many computations extremely rapidly and in parallel) to achieve our real-time system. Computer vision, considered the inverse of computer graphics, has traditionally been very well suited for GPGPU mapping, as some authors have pointed out ([12]); some have even

studied the application of pure GPGPU techniques to other particle filtering tracking algorithms with great success (like the 2D visual tracker in [13, 14]). Stream processing simplifies the scenario for every GPGPU project, but computer vision is still a clear winner due to the fact that some GPU functions (like Projective Texture Lookups [11]) remain in next generation GPUs [5] and have specific silicon devoted to them. To the best of our knowledge, this is the first 3D object tracker to be based on Stream processing.

## 2 Method Overview

### 2.1 Sparse Template Condensation Tracking

To define the problem of tracking, consider the state sequence  $\{\mathbf{x}_t, t \in \mathbb{N}\}$  of a target, composed of, in our case by:

$$\mathbf{x}_t = (T_x, T_y, S, R_x, R_y, R_z, \alpha) \quad (1)$$

states (we omit for simplicity some instances of subscript  $t$ ), where  $T_x, T_y$  are the translation coordinates of the object under study,  $S$  is the scale,  $R_x, R_y, R_z$  are the rotations along each axis and  $\alpha$  is a global illumination variable. When the target to track is a human face,  $\mathbf{x}_t$  represents the location and pose of that face (Fig. 1).

This state-space vector is given in each discrete time step  $t > 0$  by:

$$\mathbf{x}_t = \mathbf{f}_t(\mathbf{x}_{t-1}, \mathbf{v}_{t-1}) \quad (2)$$

where  $\mathbf{f}_t: \mathbb{R}^{n_x} \times \mathbb{R}^{n_v} \rightarrow \mathbb{R}^{n_x}$  is a possibly non-linear function of state  $\mathbf{x}_{t-1}$ ,  $\{\mathbf{v}_{t-1}, t \in \mathbb{N}\}$  is an independent and identically-distributed process noise sequence, and  $n_x, n_v$  are dimensions of the state (7 in our case) and process noise vectors, respectively. Equation 2 is often called the dynamics model or state transition model. The objective of tracking is to recursively estimate  $\mathbf{x}_t$  from measurements

$$\mathbf{z}_t = \mathbf{h}_t(\mathbf{x}_t, \mathbf{n}_t) \quad (3)$$

where  $\mathbf{h}_t: \mathbb{R}^{n_x} \times \mathbb{R}^{n_n} \rightarrow \mathbb{R}^{n_z}$  is a possibly non-linear function,  $\{\mathbf{n}_t, t \in \mathbb{N}\}$  is an independent and identically-distributed measurement noise sequence, and  $n_z, n_n$  are dimensions of the measurement and measurement noise vectors, respectively. Equation 3 is often called the observation model.

If we take a probabilistic approach to solve this problem, we seek filtered estimates  $\tilde{\mathbf{x}}_t$  based on the set of all available measurements  $\mathbf{z}_{1:t} = \{\mathbf{z}_i, i = 1, \dots, t\}$  up to time  $t$ , together with some degree of belief in them. Namely, we want to know the p.d.f.  $p(\mathbf{x}_t | \mathbf{z}_{1:t})$ , assuming that the initial p.d.f.  $p(\mathbf{x}_0 | \mathbf{z}_0) = p(\mathbf{x}_0)$  is known.

The optimal solution to this problem can be found if we assume certain constraints [15], but this is usually not the case. One solution, the particle filter method,

**Figure 1** Results of the simultaneous tracking of four people. The frame sequence is taken from a synthetic  $1,024 \times 768$  video, the sparse templates are composed of approximately 230 feature points, and each one is tracked using 1,000 particles.



is an approximation method that makes use of Monte Carlo simulations [2]. We now consider discrete particles forming the set:

$$(\tilde{\mathbf{X}}_{t-1}, \Pi_{t-1}) = \{(\tilde{\mathbf{x}}_{t-1}^0, \pi_{t-1}^0), \dots, (\tilde{\mathbf{x}}_{t-1}^M, \pi_{t-1}^M)\} \quad (4)$$

Every particle contains information about one possible state of the system  $\tilde{\mathbf{x}}_{t-1}^j$  and its importance weight  $\pi_{t-1}^j$ . This sample set is propagated in the Condensation algorithm as follows:

**Select** A set of  $M$  new particles  $(\tilde{\mathbf{X}}_t, \Pi_t)$  is generated, by random selection from the previous particles by using a p.d.f. according to their weights  $\Pi_{t-1}$ .

**Diffuse** A Gaussian random noise is added to each parameter of each particle in  $\tilde{\mathbf{X}}_t$ .

**Measure weight** The probability  $\Pi_t$  is measured using template matching, based on the difference error between the template (the collection of all pixels forming the face in  $t = 0$ , or some carefully selected pixels of the template if we use a sparse template [16, 17] as we describe in Section 2.3) and the input image at each time step.

The matching error,  $\varepsilon_t^j$ , is calculated based on the difference in intensity values between selected pixels in the template (feature points) and the corresponding pixels in the image at time  $t$ . A feature point in the template is projected onto the image plane based on the state vector of the particle,  $\tilde{\mathbf{x}}_t^j$ . Assuming weak-perspective projection, a point on template  $\mathbf{q} = [q_x, q_y, q_z]^T$  is projected to point  $\mathbf{p} = [p_x, p_y]^T$  on the image plane as written in

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = S \cdot \mathbf{R}_{2 \times 3} \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \quad (5)$$

$$\begin{bmatrix} \mathbf{R}_{2 \times 3} \\ \omega_1 \omega_2 \omega_3 \end{bmatrix} = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_z \quad (6)$$

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos R_x & -\sin R_x \\ 0 & \sin R_x & \cos R_x \end{bmatrix} \quad (7)$$

$$\mathbf{R}_y = \begin{bmatrix} \cos R_y & 0 & \sin R_y \\ 0 & 1 & 0 \\ -\sin R_y & 0 & \cos R_y \end{bmatrix} \quad (8)$$

$$\mathbf{R}_z = \begin{bmatrix} \cos R_z & -\sin R_z & 0 \\ \sin R_z & \cos R_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (9)$$

where  $\mathbf{R}_{2 \times 3}$  denotes the  $2 \times 3$  upper submatrix of the rotation matrix  $\mathbf{R}$ . The matching error between the template and input image is defined based on the intensity of each feature point,  $J(\mathbf{q}_m)$ ,  $m = 1, \dots, N$ , and the intensity of the corresponding projected point,  $I(\mathbf{p}_m)$ , modified by the global illumination factor  $\alpha$ . More precisely, the matching error, defined as the average matching error of  $N$  feature points, can be written as

$$\varepsilon = \frac{1}{N} \sum_{m=1}^N \rho(\kappa(J(\mathbf{q}_m), I(\mathbf{p}_m))) \quad (10)$$

$$\kappa(J, I) = \frac{\alpha \cdot I - J}{J} \quad (11)$$

$$\rho(\kappa) = \frac{\kappa^2}{\kappa^2 + 1} \quad (12)$$

where  $\kappa(J, I)$  denotes the relative residual between intensity values of the feature point  $\mathbf{q}$  and the projected point  $\mathbf{p}$ , and  $\rho(\kappa)$  represents a robust function which is used to increase the robustness against outlier pixels [17]. From the matching error for each particle,  $\varepsilon^j$ ,  $j = 1, \dots, M$ , the weight of each particle is calculated as

$$\pi^j \propto 1/\varepsilon^j \quad (13)$$

where  $\sum_{j=1}^M \pi^j = 1$ . Finally, if we rearrange the particles in descending order of weight, we can estimate the current state as the average of the best  $M' \leq M$  particles by:

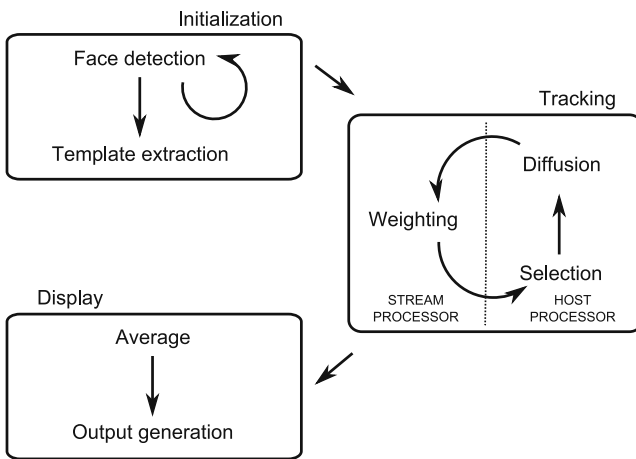
$$\tilde{\mathbf{x}}_t = \frac{\sum_{i=1}^{M'} \pi_t^i \cdot \tilde{\mathbf{x}}_t^i}{\sum_{i=1}^{M'} \pi_t^i} \quad (14)$$

## 2.2 Algorithmic Scheme

Our tracker can be studied in three big blocks: initialization, tracking and display (Fig. 2). In our system, the initialization stage is performed in a separate thread in the host system, while the main thread performs the tracking using the GPU and displays the results:

**Initialization** The initialization thread scans the image looking for new faces that are not currently being tracked. This can be done either continuously or every  $n$  frames, as new faces are not expected to pop up at any time. When a new frontal face is detected (time  $t_0$  with regard to tracking), the description template is obtained from the image.  $N$  feature points are extracted by the selected criteria, and the resulting stream of data (consisting of gray level values,  $x$ ,  $y$ ,  $z$  coordinates of the points, and normals to the face surface. See Section 2.3)





**Figure 2** Algorithmic scheme of the system.

is created for each object. At time 0,  $M$  particles per object being tracked are randomly selected and their state-space values filled with random values (uniformly distributed around the well-known initial state in  $t_0$ : the translation equals the current position of the template, the scale is 1, rotation around every angle 0, and illumination 1). The weight of every particle at this point is the same for every particle (and equals to  $1/M$ ). See Section 2.3 for a more detailed description of this stage.

**Tracking** In this stage the actual particle filtering is performed. As described before, it consists of the steps of selection or drifting, diffusion, and weight measurement. See Section 2.4 for a more detailed description of the weighting step and how the workload is divided between the host processor and the stream processor. Once the weights of each particle are obtained, we pass the result to the display stage and perform the selection of new particles, and the random diffusion of them (by means of a Gaussian noise with mean and standard deviation chosen carefully to provide both stable and fast tracking) in order to diversify the set and avoid the degeneracy [15] problem. After that, this stage starts again and is performed indefinitely, unless the quality of the tracking results degrades so much that we must consider that tracking has failed (this may be caused by the face leaving the video sequence, excessive occlusion, or if some other limits are surpassed).

**Display** After every tracking step, we average the best particles to get our approximation as the system output. The resulting state-space values can be written as an output text-file, feed to another system for consumption, or displayed on the screen as rotated, scaled and translated 3D-mesh models (Fig. 1).

### 2.3 Initialization Stage

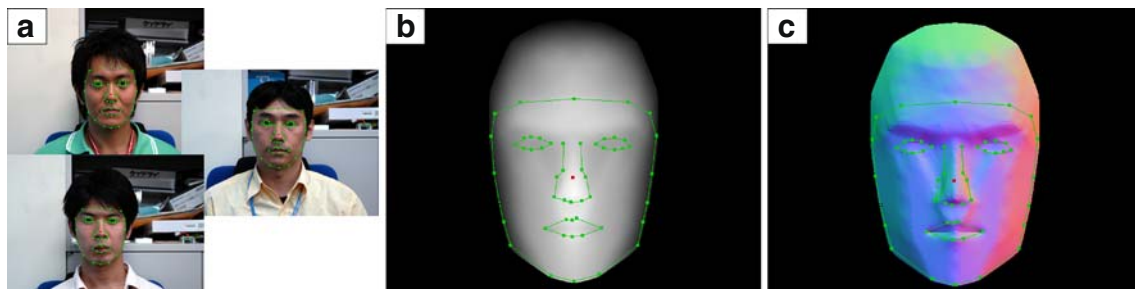
In order to detect new frontal faces in the image, a Viola and Jones [18, 19] boosting algorithm is employed, as these detectors are quite fast and have been proven to work very well in practice. After checking that the detected face is not currently being tracked (by simply comparing it to known face positions), the subimage formed by the detected rectangle is passed to the next step, for template extraction.

In the sparse template matching method [16, 17], a sparse template is carefully made up by a small set of pixels (feature points) from a full template, with the idea of making the tracking more efficient by reducing the number of calculations. In addition to this, we resort to the sparse template matching method to find relevant points to track, and treat those points as uni-dimensional streams of data to raise processing efficiency.

In this method, the human face is typically approximated as a planar surface forming the sparse template (this is, the  $z$  coordinate of each feature point is simply a constant). In order to increase the precision, more detailed models of the human face can be used: parameterized geometric models like, for example, sections of a cylinder or ellipsoid, 3D-mesh models.... Our system adopts a generic 3D model of the human face, that we personalize to each detected face by means of the Active Appearance Model.

Active Appearance Model [20] (AAM) is a Computer Vision algorithm for matching a statistical deformable model of the object's shape and appearance to a new image. They are built during a training phase using a set of images together with the coordinates of some landmarks (Fig. 3a). We perform the fitting of one of these models over the face subimage, thus obtaining the 2D coordinates of a series of landmark points (53 in our case) that correspond to previously known features of the human face (corners of the mouth, eyes, head oval, etc....). We have modified these landmarks to also include a pair of texture coordinates, a set of numbers that refer to a unique part of a texture resource stored in memory. In this way, we can create two image buffers the same size as the face image by applying (mapping) to them our 3D face model; this pair is described as a "heightmap" and a "normalmap".

A heightmap, or heightfield, is a raster image (stored as gray level image) in which each pixel represent the surface elevation data of an object. On the other hand, a normalmap, or bumpmap, is a raster image (stored as a RGB color image) in which each pixel contains a normal XYZ vector (each color component containing a coordinate). These normals represent the vector



**Figure 3** **a** Some of the images used to train the AAM, together with their landmarks; **b** the heightmap used for our generic face model; **c** the normalmap.

perpendicular to the surface of the 3D face model. See Fig. 3 for a graphical explanation of these textures, and Fig. 4 for the model creation process.

Next, the feature points are selected from local minimum/maximum points and boundary dipoles on the image, as in [16]. The local minimum/maximum points are defined as local extreme points over the gray-level distribution of the image. First, the candidate extreme points are detected by checking the 8-neighbors of each pixel in the image. The final local minimum/maximum points are then selected in ascending/descending order of the gray-level values of the candidate pixels; each point is selected so that it keeps a certain distance from other points that have already been selected. On the other hand, the boundary dipoles are defined as line segments that straddle and intersect at right angles the image boundaries in the input image. The boundary-dipole-based feature points are defined as both end points of the boundary dipoles. Here, the image boundaries are extracted as zero-cross boundaries in a zero-cross image, which is created by applying a Laplacian–Gaussian filter to the input image.

The boundary dipoles are selected in descending order of the absolute difference in gray-level values of the end points of the dipoles, where the same separation criteria is applied as in the case of minimum/maximum points.

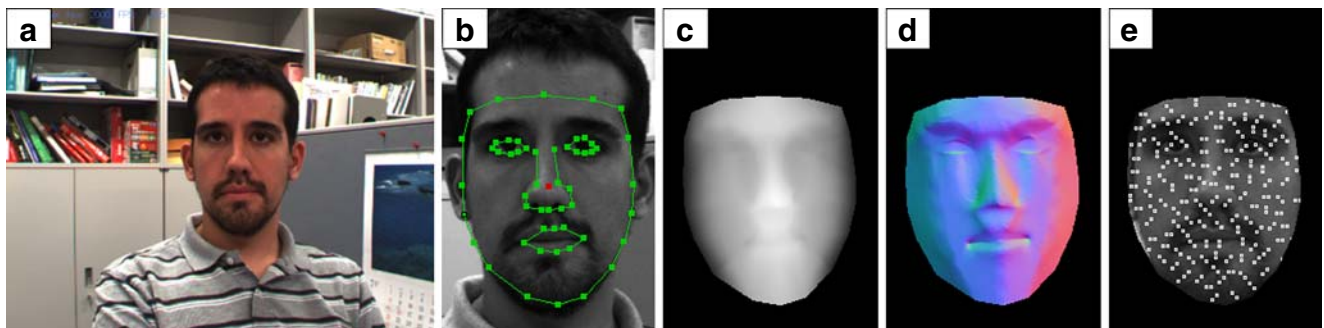
Finally, with the information taken from the three images (the face rectangular subimage, and the depth and normal buffers), our template can be formed by a stream of feature points formed as follows:

$$\mathbf{p} = (P_x, P_y, P_z, N_x, N_y, N_z, J) \quad (15)$$

where  $P_x, P_y, P_z$  are the coordinates of the feature point,  $N_x, N_y, N_z$  form the vector normal to the face surface in the feature point (see Section 2.4 for the use of this information) and  $J$  is the gray level of the feature point.

## 2.4 Weighting Stage

Of all the steps that form a PF algorithm (particle weighting, selection and diffusion), particle weight



**Figure 4** **a** Frontal face image at time  $t = 0$ ; **b** the Viola and Jones detector finds a rectangle containing a face, and the AAM is 2D-fitted to the shape of that face; **c** depth map texture is warped to the AAM shape; **d** the same for the normal map; **e** the feature

points (position and gray level) that form the sparse template are selected using image processing techniques, and their depth and normal to the surface values are extracted from prior maps.

computation is the most expensive one. What this stage does is score each particle by means of a likelihood weight between each one of the transformed feature points and the equivalent point in the current frame. The particle filter algorithm itself is computationally expensive, but the weight computation is the main bottleneck, and so we decided to subject it to Stream processing. Our two input streams are composed by the stream formed by all the particles, and the stream formed by all the feature points.

Weight calculation of each particle is an independent process, as is the matching error calculation for each feature point. Our method exploits these independencies to enable parallel processing: the kernels must perform the 3D transformation of each feature point as estimated by each of the particles, and then a comparison is made of the feature point gray level against the resulting point in the full image. The sum of all those comparisons for each feature point results in the weight of one particle. This is our output stream: the collection of weight values of every particle.

Using GPGPU techniques, we would perform this process using two passes, each one with a different kernel. The first kernel would take as input the two streams of data ( $M$  particles and  $N$  feature points) and output a 2D stream ( $M$  rows and  $N$  columns) containing in each element the error contributions of each particle and each feature point (geometrical transformation as in Eq. 5 and individual contributions as in Eq. 10). The second kernel computes a smaller stream from the larger 2D input stream, by summing all the elements in the same row (as in Eq. 10). This type of computation is called a reduction [21]. This is necessary because, in the traditional GPGPU, each processing unit is limited to its own memory and has access to only a few fast registers, unlike the much larger memory limits on a standard streaming process. CUDA, however, provides an architecture in which groups of SIMD processors (called Multiprocessors) share access to a fast, common memory. Each Multiprocessor can then be considered a standard streaming unit, and we can use that ability to perform, in just one kernel, the full weighting operation (each Multiprocessor takes one element from the stream of particles and the full  $N$  elements from the stream of feature points, and computes one weight value towards the final output stream).

We still haven't explained the use of the vectors normal to the surface in each feature point: we can employ this extra information to realize a simple feature point occlusion detector. Transforming (the same as is done with the feature positions, except we consider only the rotation effect) the normals yields a coarse measure of the 3D face model pose, that is used to discard

those points that are likely to be occluded by the face itself (that is, normal vectors that after rotation show a negative value of the  $z$  coordinate do not point towards the camera and are likely to be occluded in the final image, not contributing to the error measurement or the final weight of the particle).

### 3 CUDA Implementation Details

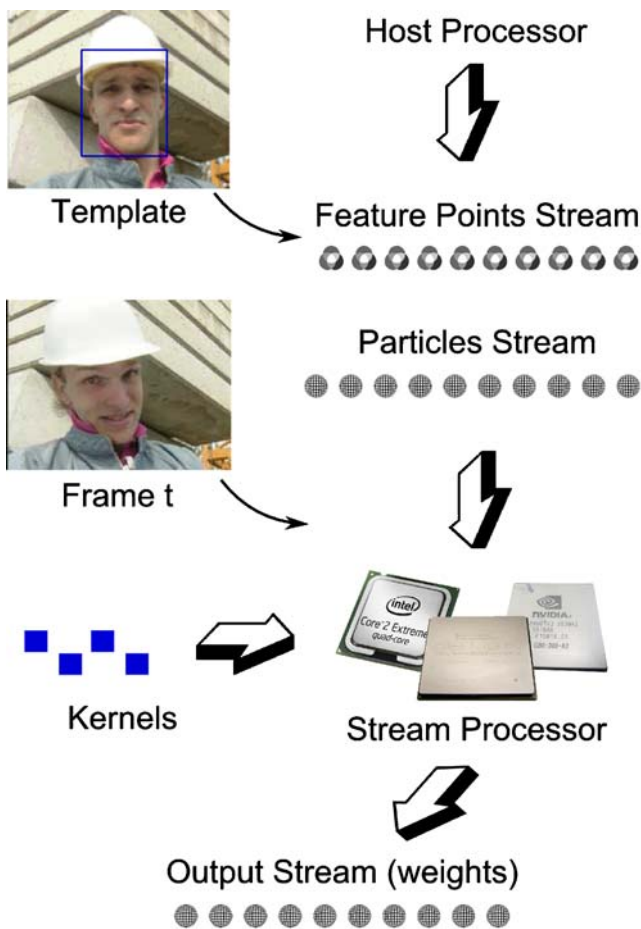
NVIDIA CUDA (Compute Unified Device Architecture [5]) is a hardware and software architecture that allows the GPU to be viewed as a data-parallel computing device that operates as a coprocessor to the main CPU (the host). The device maintains its own DRAM, that can be filled with data from the host DRAM via DMA transfers. The computing approach is that of the Stream processing paradigm: the user defines input streams and a kernel or program composed of operations to be executed over that data in a parallel fashion. In CUDA, the same kernel is executed by many different "threads" organized as follows:

A thread block is a batch of threads that can cooperate by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. Blocks of the same dimensionality and size that execute the same kernel can also be batched together (a grid of blocks), augmenting the total number of threads that can be launched by a single kernel invocation; note that there cannot be any communication between threads of different blocks.

The device is implemented as a set of multiprocessors, and each multiprocessor is composed of many processing units organized as a Single Instruction, Multiple Data architecture (SIMD). A grid is executed on the device by executing one or more blocks on each multiprocessor using time slicing: each block is split into SIMD groups of threads called warps, with a thread scheduler periodically switching from one warp to another to maximize the use of the multiprocessor's computational resources. The blocks will run sequentially if the underlying hardware has very few parallel capabilities, in parallel if it has a lot of them, or in a mixture of both.

Our algorithm (Fig. 5) uses CUDA as follows:

- At time 0, the host dumps the contents of a buffer containing the feature point stream to device memory. This input stream doesn't change during the tracking process, so it is kept in device memory and used in every iteration of the algorithm.



**Figure 5** Streams and kernels in the system.

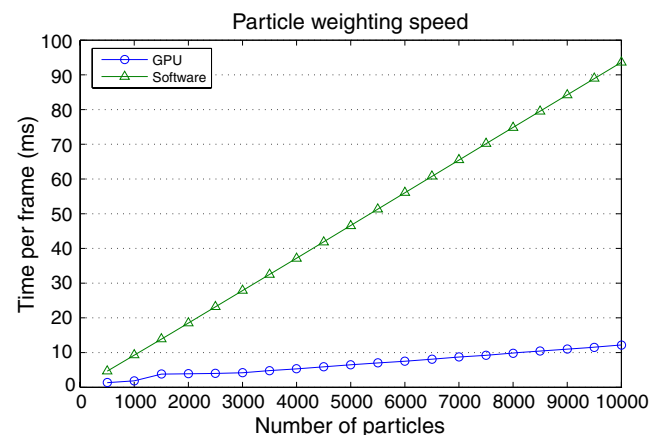
- At every step, the host sends the current video frame and a buffer containing the particle stream to device memory.
- The kernel is invoked and executed in  $M$  blocks, each with  $N$  threads (that is, one block per particle and one thread per feature point per block). Each thread computes the matching error of a particle/feature pair (transform the feature point position and normal vector, accesses the video frame pixel in that position if there is no occlusion, and compares to the feature point gray level). The result is placed by every thread in a different position of the block's shared memory.
- Next, an instruction in the kernel synchronizes all the threads in a block (makes them wait until all the threads in the block arrive to the execution point in the kernel), and then allows the first thread of every block to loop through the shared memory to sum all matching errors. The result is placed in global device memory.
- The host recovers from device memory the output stream containing the weight of every particle. The

algorithm continues its normal flow of execution until a new video frame and stream of particles requires processing.

#### 4 Experiments, Results and Future Improvement

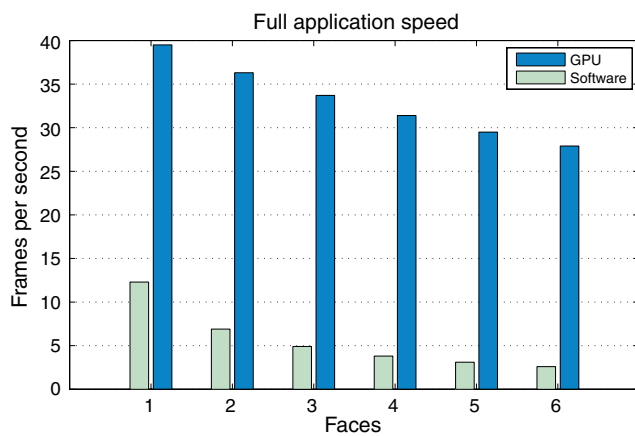
A first proof of concept was developed using the Brook language [22] from Stanford University. Brook for GPUs is a compiler and runtime implementation of the Brook stream program language for graphic chips; it quite successfully hides the burden of handling GPGPU techniques from the programmer. The algorithm was validated, but some factors prompted us to develop a more tuned version using NVIDIA CUDA [5]. While Brook works with a big range of graphic cards, CUDA is limited to just a few new architectures by NVIDIA (at the time of writing this article only GeForce 8-series GPUs) that better resemble a pure stream processor (shared memory model, scatter operations...) allowing a more efficient implementation of the algorithm.

The main limiting factor we found with Brook was the impossibility of implementing the operations described in Section 3 with only one kernel, requiring instead the use of multipass techniques (several chained kernels, each one running on the output stream of the previous one). The reason was that Brook (targeting older GPU architectures than CUDA) lacks the shared memory between processing units (and therefore, there is no possible communications between them) and also the ability of synchronizing the different threads: see Section 2.4 for a description of the steps involved in this GPGPU (as opposed to SP) technique. Brook also suffers from the excessive overhead associated with



**Figure 6** Speed of the particle weighting stage, comparing Stream processing in the GPU version to a serial CPU-only version. Video  $1,024 \times 768$ , 217 feature points.





**Figure 7** Speed of the full application, comparing stream processing in the GPU version to a serial CPU-only version. Video  $1,024 \times 768$ ,  $\approx 230$  feature points per face, and 1,000 particles per face.

using a high level language that “metacompiles” [22] to C++ and a run-time library for OpenGL/DirectX plus Cg/HLSL operations; CUDA, on the other hand, is a native architecture.

The developed software (a mixture of C++ and CUDA) has been tested on an Intel Core II 2.66 GHz host system with 2GB RAM, using a NVIDIA GeForce 8800GTX GPU as the stream processor. This GPU features 128 processing units organized in 16 multiprocessors for a peak performance (approximate) of 350 GFLOPS under CUDA. For comparison purposes, a software-only version of the weighting algorithm was also developed (single-threaded, no SIMD extensions used) for execution on the host CPU alone.

For the adaptive models, we used AAM-API, a C++ implementation of the Active Appearance Model framework developed by Mikkel B. Stegmann [23]. The generic face model heightmap and normalmap were created by hand on the base of a subdivided CANDIDE [24], a parameterized face mask specifically developed for model-based coding of human faces. The Viola and Jones implementation provided with OpenCV (Open Computer Vision Library) was used as the frontal face detector.

The results indicate an important speed boost compared to the CPU-only version of the algorithm, especially when using a large number of particles (Fig. 6) and/or tracking multiple objects simultaneously (Fig. 7), making the tracker eminently suitable for real-time processing in a standard PC platform.

Unlike other approaches to fast hardware solutions for particle filtering, like dedicated architectures or FPGA solutions, stream processing targets commercial off-the-shelf (COTS) processors, be it a GPU, a Cell

BE, or a Symmetric Multi-Processor (SMP). However, many things can be learned from these approaches (usually taken with the inherent parallelism of PFs in mind) that could lead to big improvements in our SP implementation.

One of those papers is [25], in which considerable effort was made to improve resampling algorithms for PFs so that they could be efficiently implemented in hardware in a distributed fashion. In our specific PF problem, we feel the weighting/importance step represents a bigger bottleneck than the selection/resampling step, so we have decided to perform the latter on the CPU (we are thus performing Centralized Resampling in [25] terminology). Other types of distributed resampling could lead to the complete tracking stage being performed on the GPU, without having to move all weight floating point values from device to host memory and thus overcoming the graphic bus bandwidth bottleneck.

## 5 Conclusions

We have described a system for 3D visual tracking capable of achieving real-time performance thanks to the use of a GPU for parallel computation. The use of the stream processing approach greatly simplified the development issues, and at the same time opened the door to other computing architectures. The goals imposed before starting the design (automatic, robust, just one camera, conventional computing resources, multi-object, real-time) have been all achieved, and the system is currently being used for future research in the area of conversation scene analysis [1] as expected. The novelty of the proposed work lays not only in the usage of a stream processor for 3D visual tracking, but also in an improved sparse template initialization method that improves the accuracy and stability of tracking by means of a simple, generic 3D-model of the human face.

Anyone observing the trends in processors’ transistor counts and computing power will notice that, in order to continue achieving performance improvements, processor vendors have shifted their strategy from increasing clock speed to increasing the number of cores per processor. Software has, therefore, to keep up with this concurrency drive. Multimedia applications in particular are hungry for new computing power, and their special characteristics mean that future architectures will differ from the ones we are used to. Stream processors are a strong candidate to fulfill that demand (modern GPUs offer incredible amounts of raw processing power). Fortunately, the paradigm driving the new hardware can meet this challenge (exposing

desirable properties of multimedia algorithms, as well as creating natural ways of implementing them, as in our particle filtering case).

**Open Access** This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

## References

- Otsuka, K., Yamato, J., Takemae, Y., & Murase, H. (2006). Conversation scene analysis with dynamic Bayesian network based on visual head tracking. In *Proc. IEEE intl. conf. on multimedia and expo 2006* (pp. 949–952).
- Doucet, A., Freitas, N., & Gordon (Eds.) (2001). *Sequential Monte Carlo methods in practice*. Springer.
- Isard, M., & Blake, A. (1998). Condensation—conditional density propagation for visual tracking. In *Proc. of the 6th intl. conf. on computer vision* (pp. 107–112).
- Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., et al. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1), 80–113.
- NVIDIA (2007). CUDA (compute unified device architecture) programming guide ver.1.0. <http://developer.nvidia.com/object/cuda.html>.
- Peercy, M., Segal, M., & Gertsman, D. (2006). A performance-oriented data parallel virtual machine for GPUs. In *Proc. SIGGRAPH 2006*.
- McCool, M. (2007). Multi-core cpus, accelerators, and the many-core future: A unified software approach with rapid-mind. In *Proc. SIGGRAPH 2007*.
- Kapasi, U. J., Rixner, S., Dally, W. J., Khailany, B., Ahn, J. H., Mattson, P., et al. (2003). Programmable stream processors. *IEEE Computer*, 36, 54–62, August.
- Venkatasubramanian, S. (2003). The graphics card as a stream computer. In *SIGMOD-DIMACS workshop on management and processing of data streams*.
- Kapasi, U., Dally, W. J., Rixner, S., Owens, J. D., & Khailany, B. (2002). The imagine stream processor. In *Proc. of intl. conf. on computer design* (pp. 282–288).
- Khailany, B., Williams, T., Lin, J., Long, E., Rygh, M., Tovey, D., et al. (2007). A programmable 512 gops stream processor for signal, image, and video processing. In *IEEE international solid-state circuits conference 2007 digest of technical papers* (pp. 272–602).
- Fung, J., & Mann, S. (2004). Computer vision signal processing on graphics processing units. In *Proc. IEEE international conference on acoustics, speech, and signal processing* (Vol. 5, pp. 93–96).
- Montemayor, A. S., Pantrigo, J. J., Sánchez, Á., & Fernández, F. (2004). Particle filter on gpus for real-time tracking. In *Proc. of ACM SIGGRAPH* (p. 94).
- Montemayor, A. S., Pantrigo, J. J., Cabido, R., & Payne, B. (2006). Bandwidth improved GPU particle filter for visual tracking. In *Ibero-American symposium on computer graphics—SIACG*.
- Arulampalam, S., Maskell, S., Gordon, N. J., & Clapp, T. (2002). A tutorial on particle filters for on-line non-linear/non-Gaussian Bayesian tracking. *IEEE Transactions of Signal Processing*, 50(2), 174–188, February.
- Matsubara, Y., & Shakunaga, T. (2005). Sparse template matching and its application to real-time object tracking. *IPSI Transactions on Computer Vision and Image Media*, 46(9), 17–40.
- Matsubara, Y., & Shakunaga, T. (2004). Real-time object tracking by sparse template matching. In *IPSI SIG technical report*, no. 26 (pp. 49–56).
- Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Proc. of the IEEE computer society conference on computer vision and pattern recognition* (Vol. 1, pp. 511–518).
- Viola, P., & Jones, M. (2004). Robust real-time face detection. *International Journal of Computer Vision*, 57(2), 137–154.
- Edwards, G. J., Taylor, C. J., & Cootes, T. F. (1998). Interpreting face images using active appearance models. In *Proc. intl. conf. on face and gesture recognition* (pp. 300–305).
- Horn, D. (2005). *Stream reduction operations for GPGPU applications*. GPU Gems 2. Addison Wesley.
- Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., et al. (2004). Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3), 777–786.
- Stegmann, M. B., Ersbøll, B. K., & Larsen, R. (2003). Fame - a flexible appearance modelling environment. *IEEE Transactions on Medical Imaging*, 22(10), 1319–1331.
- Ahlberg, J. (2001). Candide-3 – an updated parameterized face. Technical report, Dept. of Electrical Engineering, Linköping University.
- Bolić, M., Djurić, P. M., & Hong, S. (2005). Resampling algorithms and architectures for distributed particle filters. *IEEE Transactions on Signal Processing*, 53(7), 2442–2450, July.



**Oscar Mateo Lozano** received the B.Sc. and M.Sc. degrees in Electrical Engineering from the Universidad Politécnica de Valencia in Spain, in 2004. Currently, he is a Ph.D. candidate at the Image Processing Group of the Universidad Politécnica de Madrid in Spain. He has researched medical image processing in Brazil, sensor networks in the USA, computer graphics in Spain, video processing in Japan, and has also strong industry “hands-on” experience with networking and embedded architectures. His current main research interests are in the areas of computer graphics and high-performance computing architectures.



**Kazuhiro Otsuka** received his B.E. and M.E. degrees in electrical and computer engineering from Yokohama National University in 1993 and 1995, respectively. He joined the NTT Human Interface Laboratories in 1995. He received his Ph.D. in information science from Nagoya University in 2007. He is now a senior research scientist in the NTT Communication Science Laboratories. His current research interests include computer vision and communication scene analysis. He received the Best Paper Award of IPSJ National Convention in 1998 and the Best Paper Award of IAPR International Conference on Image Analysis and Processing in 1999. He is a member of the IEEE, the IPSJ, and the IEICE.